XSEarch: A Semantic Search Engine for XML

Sara Cohen Jonathan Mamou Yaron Kanza Yehoshua Sagiv

School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
{sarina, mamou, yarok, sagiv}@cs.huji.ac.il

Abstract

XSEarch, a semantic search engine for XML, is presented. XSEarch has a simple query language, suitable for a naive user. It returns semantically related document fragments that satisfy the user's query. Query answers are ranked using extended information-retrieval techniques and are generated in an order similar to the ranking. Advanced indexing techniques were developed to facilitate efficient implementation of XSEarch. The performance of the different techniques as well as the recall and the precision were measured experimentally. These experiments indicate that XSEarch is efficient, scalable and ranks quality results highly.

1 Introduction

It is becoming increasingly popular to publish data on the Web in the form of XML documents. Current search engines, which are an indispensable tool for finding HTML documents, have two main drawbacks when it comes to searching for XML documents. First, it is not possible to pose queries that explicitly refer to meta-data (i.e., XML tags). Hence, it is difficult, and sometimes even impossible, to formulate a search query that incorporates semantic knowledge in a clear and precise way.

The second drawback is that search engines return references (i.e., links) to documents and not to specific fragments thereof. This is problematic, since large XML documents (e.g., the XML DBLP) may contain thousands of elements storing many pieces of information that are not necessarily related to each other. For example, an author is related to titles of papers she wrote, but not to titles of other papers. Actually, if a search engine simply matches the search terms against the documents, it may return documents that do not answer the user's query. This occurs when distinct search terms are matched to unrelated parts of an XML document, as illustrated in the next example.

Example 1.1 Suppose that a user is trying to find papers of Vianu on the topic of logical databases. This might be formulated as the search query: “Vianu logical databases”. Consider the XML document fragment below, an excerpt from the XML version of DBLP.

```
<proceedings>
  <inproceedings>
    <author>Moshe Y. Vardi</author>
    <title>Querying Logical Databases</title>
  </inproceedings>
  <inproceedings>
    <author>Victor Vianu</author>
    <title>A Web Odyssey: From Codd to
           XML</title>
  </inproceedings>
</proceedings>
```

A standard search engine would regard the document above as an appropriate response, since it mentions all the search terms. However, we can easily see that although the search terms appear, they do not all appear in the same context. Thus, the document is not an ideal response to the user's query. The problem arises since the document is treated as an integral unit.

Since a reference to a whole XML document is usually not a useful answer, the granularity of the search should be refined. Instead of returning entire documents, an XML search engine should return fragments of XML documents.

A query language for XML, such as XQuery, can be used to extract data from XML documents. However, such a query language is not an alternative to

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

an XML search engine for several reasons. First, the syntax of XQuery is by far more complicated than the syntax of a standard search query. Hence, it is not appropriate for a naive user. Second, rather extensive knowledge of the document structure is required in order to correctly formulate a query. Thus, queries must be formulated on a per document basis. Finally, XQuery lacks any mechanism for ranking answers—an essential feature, since there are likely to be many answers when querying large XML documents.

There have been several attempts to extend XQuery-like languages with information-retrieval techniques [5, 10, 11, 6, 15]. However, those languages still suffer from a complex query syntax. Another approach is to add capabilities of meta-data querying to search engines [4, 2]. But answers to those search engines are not required to consist of semantically-related pieces of information, and thus suffer from the problem illustrated in Example 1.1.

In [9], we have investigated under what conditions different elements of an XML document are semantically related. In this paper, we show how the theoretical results of [9] can be efficiently combined with information-retrieval techniques to yield XSEarch—a search engine for XML. The design and implementation of XSEarch involved several challenges. First, we developed a syntax for search queries that is suitable for a naive user and facilitates a fine-granularity search. The syntax allows, but does not require, the user to specify how keywords are related to tags; in fact, a search query may consist only of keywords. Second, the theoretical results of [9] were adapted so that XSEarch always returns, as answers, document fragments that are semantically related, even when only keywords (and no tags) are specified in the query. Third, we have combined the notion of semantic relationship with traditional information-retrieval techniques to guarantee that answers are not merely semantically-related fragments, but actually fragments that are highly relevant to the keywords of the query. Fourth, we developed a suitable ranking mechanism that takes into account both the degree of the semantic relationship and the relevance of the keywords. Fifth, we developed index structures and evaluation algorithms that allow the system to deal efficiently with large documents, containing thousands of kilobytes of information, and to generate answers in an order similar to their ranking (thus, avoiding the overhead of sorting all answers before returning any). Sixth, the implementation of XSEarch is *extensible* in the sense that it can easily accommodate different types of semantic relationships.

Section 2 describes the syntax of search queries. Section 3 presents the semantics of queries, which is based on the theory developed in [9]. In Section 4, we show how to rank answers by extending information-retrieval techniques. In Section 5, the XSEarch system implementation is presented. In Section 6, we present our experimental results. Finally, Section 7 considers related work and then concludes.

2 Query Syntax

The query language of a standard search engine is simply a list of keywords. In some search engines, each keyword can optionally be prepended by a plus sign ("+"). Keywords with a plus sign *must* appear in a satisfying document, whereas keywords without a plus sign *may* or *may not* appear in a satisfying document (but the appearance of such keywords is desirable).

The query language of XSEarch is a simple extension of the language described above. In addition to specifying keywords, we allow the user to specify labels and keyword-label combinations that must or may appear in a satisfying document.

Formally, a *search term* has the form $l\!:\!k$, $l\!:$ or $:\!k$ where l is a label and k is a keyword. A search term may have a plus sign prepended, in which case it is a *required* term. Otherwise, it is an *optional* term. We use t, t_1, t_2, etc., as an abstract notation for required and optional terms. A *query* has the form $Q(S)$ where $S = t_1, \ldots, t_m$ is a sequence of required and optional search terms. We sometimes refer to the above query as Q, when S is clear from the context.

3 Query Semantics

This section presents the semantics of our queries. In order to satisfy a query Q, each of the required terms in Q must be satisfied. In addition, the elements satisfying Q must be *meaningfully related*. However, it is difficult to determine when a set of elements is meaningfully related. Therefore, we assume that there is a given relationship R that determines when two nodes are related. We then show how to extend R for arbitrary sets of nodes. We also give one natural example of a relationship, which we call *interconnection*. In our working system we use the interconnection relationship. However, it is possible to use a different relationship, with little impact on system efficiency.

3.1 Satisfaction of a Search Term

We model XML documents as trees in the standard fashion.[1] Each *interior node* is associated with a label and each *leaf node* is associated with a sequence of keywords. If k is a keyword in the sequence associated with n, we will also say that n *contains* k.

In Figure 1 there is a tree that represents a small portion of the XML document of the Sigmod Record.

[1]XML is sometimes modeled as a graph, instead of a tree, by taking ID/IDREF and XLink links into consideration. In principle, it is possible to use our system even when XML is modeled as a graph. To simplify, we only consider XML trees in this paper.

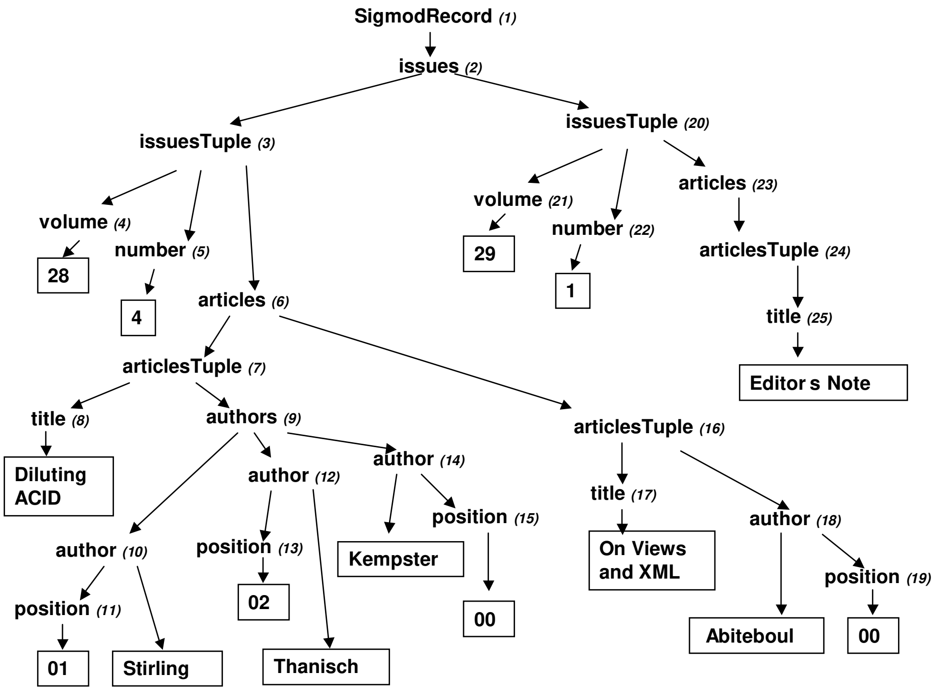

Figure 1: Part of the Sigmod Record document tree.

We will refer to this tree as $\mathcal{T}_{sr}$. The interior nodes are numbered to allow easy reference.

Let n be an interior node in a tree T. We say that n satisfies the search term $l\!:\!k$ if n is labeled with l and a descendent that contains the keyword k. We say that n satisfies the search term $l\!:$ if n is labeled with l. Finally, we say that n satisfies the search term $:\!k$ if n has a leaf child that contains the keyword k.

Example 3.1 In the tree $\mathcal{T}_{sr}$, node number 14 satisfies `:Kempster` and node number 9 satisfies `authors:Kempster`. However, node 9 does not satisfy `:Kempster`, `position:` or `:position`.

3.2 Meaningfully Related Sets of Nodes

Let T be a tree and R be a binary, reflexive and symmetric relationship on the nodes in T. We assume that R contains pairs of nodes that are meaningfully related. We present two different ways to extend R to arbitrary sets of nodes.

We say that a set of nodes N is *all-pairs R-related*, denoted $\approx_a^R \{N\}$, if (n_1, n_2) is in R, for every pair of nodes n_1, n_2. Intuitively, this states that a set of nodes is meaningfully related if every pair of nodes in the set is meaningfully related. We say that N is *star R-related*, denoted $\approx_s^R\{N\}$, if there is a node $n_* \in N$ such that the pair (n_*, n) is in R, for all nodes $n \in N$. We call n_* the *star center*. Intuitively, this states that the nodes of a set are meaningfully related if all these nodes are meaningfully related to a node in the set.

Depending on the structure of the documents in a corpus, either the all-pairs relationship or star relationship may be more appropriate. This will be discussed in detail later on in Section 3.6.

3.3 Query Answers

Let $Q(t_1, \ldots, t_m)$ be a query. We say that a sequence $N = n_1, \ldots, n_m$ of nodes and null values is an *all-pairs R-answer* for Q if the nodes in N are all-pairs R-related and for all $1 \leq i \leq m$:

1. n_i is not the null value if t_i is a required term;
2. n_i satisfies t_i if it is not the null value.

Similarly, N is a *star R-answer*, when the nodes in N are star R-related.

We use $Ans_T^{a,R}(Q)$ to denote the set of all-pairs R-answers for the query Q over a tree T and use $Ans_T^{s,R}(Q)$ to denote the set of star R-answers for Q over T. It is not difficult to see that for all trees T, relationships R and queries Q, $Ans_T^{a,R}(Q) \subseteq Ans_T^{s,R}(Q)$. Actually, if R is a transitive relationship then, $Ans_T^{a,R}(Q) = Ans_T^{s,R}(Q)$.

Our query answers can have null values in their sequences. However, we are interested in answers that have maximal information. Let $Q(S)$ be a query and let both N and N' be either all-pairs R-answers or star R-answers. We say that an answer N' *subsumes* N if N' is equal to N on all non-null values of N. Intuitively, if N' subsumes N, then it contains more information. We say that an answer N is *maximal* if every answer that subsumes N is actually equal to N.

For $\square \in \{a, s\}$, we use $MaxAns_T^{\square,R}(Q)$ to denote the set of maximal answers in $Ans_T^{\square,R}(Q)$.

3.4 The Interconnection Relationship

We present a relation which can be used to determine whether a pair of nodes is meaningfully related. We found this relation to be intuitive, and it gave appropriate results in our working system.

Let T be a tree and let n_1 and n_2 be nodes in T. The shortest undirected path between n_1 and n_2 consists of the paths from the lowest common ancestor of n_1 and n_2 to n_1 and n_2. We denote the tree consisting of these two paths as $T_{|n_1,n_2}$. Intuitively, this tree describes the relationship between the nodes n_1 and n_2. For example in $\mathcal{T}_{sr}$, depicted in Figure 1, the tree $T_{|8,13}$ consists of the nodes 7, 8, 9, 12 and 13.

We start by giving an intuitive understanding of relationships in a document tree. One may view a node in a tree as representing an entity in the world. Two different nodes with the same label correspond to different entities of the same type. If n_a is an ancestor of n, then we may understand that n belongs to the entity that n_a represents. Now, suppose that nodes n and n' have distinct ancestors n_a and n'_a, respectively, such that n_a and n'_a have the same label. Suppose also that n'_a is not an ancestor of n, and n_a is not an ancestor of n'. We may conclude that n and n' are not meaningfully related since they belong to different entities of the same type. Note that both n_a and n'_a must be in the relationship tree of n and n'. Otherwise, they would be ancestors of both n and n' and would not imply that n and n' are unrelated.

We demonstrate and extend this intuition with a few examples. Consider nodes 4 and 5 in $\mathcal{T}_{sr}$ (Figure 1). Their relationship tree does not contain two nodes with the same label. Therefore, nodes 4 and 5 are related. However, nodes 4 and 22 are not related since their relationship tree contains different nodes with the label `issuesTuple`. This reflects the intuition that 4 is the `volume` of the issue with `number` node 5 and not the `volume` of the issue with `number` node 22. Now, consider nodes 11 and 12 in $\mathcal{T}_{sr}$. Node 11 belongs to the `author` node numbered 10. However, 12 is a different `author` node. Therefore, we may conclude that the `position` in node 11 belongs to node 10 and is not related to node 12. As a final example, consider nodes 10 and 12. These nodes share the same label. However, all their ancestors are the same, and thus, they belong to the same entities. Therefore, we may conclude that nodes 10 and 12 are meaningfully related. In fact, nodes 10 and 12 represent different authors, but they are related by virtue of belonging to the same article.

We formalize this idea. Let n and n' be nodes in T. We say that n and n' are *interconnected* if one of the following conditions holds:

1. $T_{|n,n'}$ does not contain two distinct nodes with the same label *or*
2. the only two distinct nodes in $T_{|n,n'}$ with the same label are n and n'.

We use R_i to denote the interconnection relationship. In the sequel, only the interconnection relationship between nodes will be considered. Hence, usually we will not specify explicitly the relationship considered.

3.5 Complexity

In combined complexity both the document and the query are considered as part of the input. This is often of interest since queries may be quite large. In input-output complexity, we analyze the complexity of a problem as a function of the input (i.e., query and document) and the output. This complexity measure is useful when both queries and query results are large. The following complexity results are from [9].

Theorem 3.2 (Evaluation Complexity) *Let T be a tree and let $Q(S)$ be a query.*

- *Determining whether $MaxAns_T^{a,R_i}(Q) \neq \emptyset$ is NP-complete under combined complexity.*
- *If S contains only optional terms, then $MaxAns_T^{a,R_i}(Q)$ can be computed in polynomial time under input-output complexity.*
- *The set $MaxAns_T^{s,R_i}(Q)$ can be computed in polynomial time under input-output complexity.*

3.6 Examples of Query Semantics

Example 3.3 Consider the query Q_1, defined as Q_1(+`title:`, `author:`). The query Q_1 finds pairs of titles and authors, belonging to the same article. Only tuples where the title is non-null will be returned. For both all-pairs and star semantics, the answers created for $\mathcal{T}_{sr}$ are the same, namely $(8, 10)$, $(8, 12)$, $(8, 14)$, $(17, 18)$ and $(25, \perp)$.

Consider also the portion of the DBLP document presented in Figure 2. The answers for Q_1 over this document would consist of $(6, 3)$ and $(6, 4)$. Observe that although this document differs in structure from $\mathcal{T}_{sr}$, the correct pairs are found for both documents.

Example 3.4 Consider the query Q_2 that looks for volumes, authors with the name `Kempster`, and authors who have published with `Kempster`: Q_2(+`volume:`, +`author:Kempster`, `author:`).

The set of maximal all-pairs answers for Q_2 over $\mathcal{T}_{sr}$ only contains answers for which both authors appear under the same `articlesTuple` node. However, the set of maximal star answers for Q_2 over $\mathcal{T}_{sr}$ also contains authors that appear in the same issue, but in different articles. Thus, star answers require a “looser”

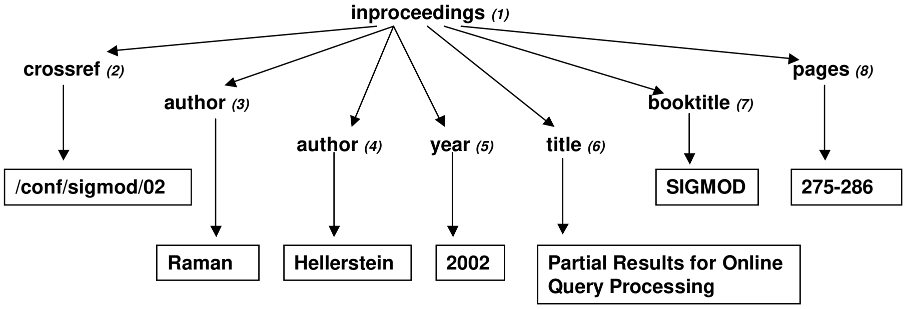

Figure 2: Part of the DBLP document tree.

relationship between the nodes in the answer, and can be viewed as an approximation for all-pairs answers.

Currently, in the Sigmod Record, `author` elements contain their `position` as an attribute and the name of the author simply as data. However, `author` elements could have been modeled slightly differently, with both `name` and `position` as subelements, e.g.,

```
<author>
   <name>Stirling</name>
   <position>01</position>
</author>
```

Observe that Q_2 would still retrieve the same answers. However, $Q_3(+\texttt{volume:},\ +\texttt{name:Kempster},\ \texttt{name:})$ would not retrieve any answer under all-pairs semantics with a non-null element in its optional-component. Under star semantics, similar answers to those of Q_2 would be retrieved.

Example 3.5 In Example 1.1 the user wished to find papers of Vianu on the topic of logical databases. In the absence of knowledge of the tags in the document, the query $Q_4(+\texttt{:Vianu},\ +\texttt{:logical},\ +\texttt{:databases})$ would be used for this purpose. When applying Q_4 to the document in Example 1.1, the answer will be empty for both all-pairs and star semantics. We get an empty result since the `title` node that matches `logical` and `databases` is not interconnected to the `author` node that matches `Vianu`. This conforms to our intuition of the meaning of Q_4.

4 Ranking Answers

In the previous section, we presented our semantics for query answers. These semantics combined database-like ideas (e.g., computing a projection of a document, clear-cut semantics) with a heuristic for defining relationships between nodes. In this section, we extend our semantics with extended traditional information retrieval techniques to rank query answers.

4.1 Weight of a Keyword and of a Label

We compute the weight of a keyword (sometimes called a *term*) k in a given leaf node n_l using a variation of the standard *tfidf* (term frequency, inverse document frequency) formula. Normally, given a document D and a keyword k appearing in D, the *tfidf* formula defines a value $\mathit{tfidf}(k, D)$ that represents both the frequency of the keyword k in the document D (i.e., the *tf*) and the inverse frequency of the keyword in all the documents in the given corpus (i.e., the *idf*).

In XSEarch, subtrees of a document are returned to the user. Hence, we compute the weight of keywords at a lower granularity, i.e., at the level of the leaf nodes of a document. This allows us to determine which subtrees of a document are more relevant, thereby enabling us to properly rank query results.

Let k be a keyword and n_l be a leaf node. We use $\mathit{occ}(k, n_l)$ to denote the number of times that k appears in n_l. The term frequency of k in n_l is defined as

$$\mathit{tf}(k, n_l) := \frac{\mathit{occ}(k, n_l)}{\max\{\mathit{occ}(k', n_l) \mid k' \in \mathit{words}(n_l)\}} .$$

This is a standard variation of the *tf* formula that gives a larger weight to frequent keywords in sparse nodes than to those in nodes with many keywords. Let N be the set of all leaf nodes in the corpus. Then, we define

$$\mathit{ilf}(k) := \log\left(1 + \frac{|N|}{|\{n' \in N \mid k \in \mathit{words}(n')\}|}\right) .$$

Intuitively, $\mathit{ilf}(k)$ is the logarithm of the inverse leaf frequency of k, i.e., the number of leaves in the corpus over the number of leaves that contain k.

Now we define $\mathit{tfilf}(k, n_l)$ as $\mathit{tf}(k, n_l) \times \mathit{ilf}(k)$. Note that by taking a log in the *ilf* factor, we increase the overall importance of the *tf* factor. In XSEarch the weight of each keyword in each node is stored in an index. The actual weight stored is a normalized version of the value $\mathit{tfilf}(k, n_l)$, denoted as $w(k, n_l)$. By definition, $w(k, n_l)$ is 0 if k does not appear in n_l. A weight of 0 is not stored explicitly in our index.

Each label l is associated with a weight $w(l)$ that determines its importance. The label weights can be either user defined or system generated. For example, the user may choose to give the label `title` a greater weight than the label `section`. As of now, if the user does not specify label weights, then the system gives the same weight to all labels. In the future we may implement other methods to automatically determine the label weight, such as giving higher weight to less common labels. This information can be derived from the indices we create.

4.2 Ranking Factors

A query can have many answers. Therefore, it is of primary importance to rank the answers by their estimated relevance. The XSEarch Ranker gives a *score* to each query answer N by taking into consideration both the structure of the result as well as its contents. The factors considered are described in detail below.

Query and Answer Similarity

We use the vector space model, common in information retrieval [1], when determining how well an answer satisfies a query. Let $\mathcal{L}$ be the set of all labels and $\mathcal{K}$ be the set of all keywords. Each interior node n in the corpus is associated with a vector V_n of size $|\mathcal{L} \times \mathcal{K}|$. The vector V_n is called the *profile* of n. The profile of n has an entry for each pair $(l, k) \in \mathcal{L} \times \mathcal{K}$. We use $V_n[l, k]$ to denote the entry of V_n corresponding to the pair (l, k). Let N_{leaf} be the set of leaf descendents of n. The values in the profile of n, (i.e., V_n) are defined as follows:

$$V_n[l,k] = \begin{cases} \sum_{n' \in N_{\text{leaf}}} w(k, n') & \text{if } \mathit{label}(n) = l \\ 0 & \text{otherwise} \end{cases}$$

Consider a search term t. (It is irrelevant for our purposes here whether t is an optional or a required term.) The search term t is associated with a vector of size $|\mathcal{L} \times \mathcal{K}|$, denoted V_t. The entries in the vector V_t are defined as follows. If t is of the form $:k$, then V_t has the value 1 in all dimensions that correspond to the keyword k, and 0 in all other dimensions. If t is of the form $l:$, then V_t has $w(l)$ in all dimensions that correspond to the label l, and 0 in all other dimensions. If t has the form $l:k$, then V_t has the value $w(l)$ in exactly the dimension corresponding to (l, k), and 0 in all other dimensions.

The *measure of similarity* between a query Q and an answer N, denoted $\mathit{sim}(Q, N)$, is the sum of the cosine distances between the vectors associated with the nodes in N and the vectors associated with the terms that they match in Q.

Relationships between Nodes

Let N be a query answer. The nodes in N may appear in the document tree in many different places. Some configurations of the nodes in N seem to be more meaningful. We use $\mathit{tsize}(N)$ to denote the number of nodes in the relationship tree of N. If this value is small, then the nodes in N are close together. Hence, they are more likely to be meaningfully related.

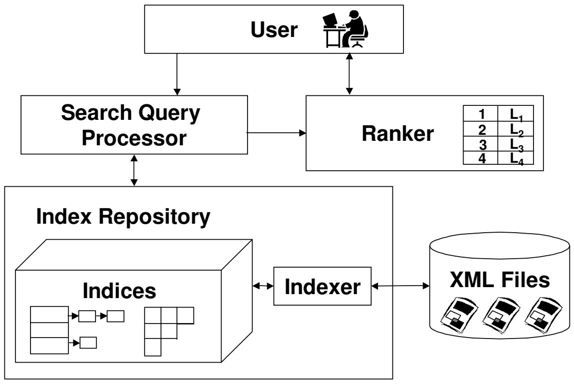

Figure 3: XSEarch system architecture.

We say that nodes n and n' participate in an *ancestor-descendent relationship* if n is the ancestor of n' or n' is the ancestor of n. In many XML documents, this type of configuration tends to indicate a strong relationship between n and n'. We define $\mathit{anc\text{-}des}(N)$ as the number of unordered pairs from N that participate in an ancestor-descendent relationship.

Combining the Factors

Given a query Q and an answer N, we use the measures $\mathit{sim}(Q, N)$, $\mathit{tsize}(N)$ and $\mathit{anc\text{-}des}(N)$ to determine the ranking of the answer. We experimented with the following combination of factors by varying the values of α, β and γ

$$\frac{\mathit{sim}(Q,N)^{\alpha}}{\mathit{tsize}(N)^{\beta}} \times \left(1 + \gamma \times \mathit{anc\text{-}des}(N)\right). \tag{1}$$

Section 6 contains additional details on our experimentation in ranking.

5 System Implementation

Our first attempt at implementing our language was to try to translate our queries to XQuery and use an XQuery processor to create the query results. This approach was found inappropriate for several reasons. It was possible to translate our queries to XQuery, since the full XQuery language has Turing machine expressive power. However, the translation was extremely complicated. We tried running queries generated by our translation on several different XQuery processors. Even for very small queries and extremely small documents (of size < 20KB), query execution took several hours. In addition, XQuery does not yet have any built-in ranking mechanism, which is sorely needed in the context of searching large document sets. Actually, it is not surprising that this implementation proved infeasible since XQuery systems are generally not optimized for queries of the type that we produced. Hence, we created our own working system, from scratch.

```
ComputeInterconnectionIndex(T)

for i:=|T|-1 down to 0 do
    for j:=i+1 to |T| do
        if i is an ancestor of j then
            let i_c be the child of i that is on the path to j
            let j_p be the parent of j
            intercon[i,j] := intercon[i_c,j] and label(i_c)≠label(j) and
                             intercon[i,j_p] and label(i)≠label(j_p)
for i:=1 to |T|-1 do
    for j:=i+1 to |T| do
        if i is not an ancestor of j then
            let i_p be the parent of i
            let j_p be the parent of j
            intercon[i,j] := intercon[i_p,j] and label(i_p)≠label(j) and
                             intercon[i,j_p] and label(i)≠label(j_p)
```

Figure 4: Computing the interconnection index using dynamic programming.

The architecture of the XSEarch system is depicted in Figure 3. The basic flow of information is as follows. The user enters a query using a browser. The Search-Query Processor parses the query into a list of search terms. The Index Repository is used to find nodes that satisfy the search terms and to find out whether pairs of nodes are interconnected. The Index Repository responds by checking the stored indices. If these indices do not contain sufficient information, as may be the case when dynamic online indexing is employed (see Section 5.2), the Indexer is used to augment the current indices. Once the relevant information is returned to the Search-Query Processor, it creates the answers, which are ranked, sorted and then returned.

We discuss the implementation of the index repository. Implementing this component efficiently was one of the main challenges for our system. The Ranker, another important component, was discussed in Section 4. The query processor is based on the algorithms presented in [9], uses the index structures presented here, and is not discussed due to lack of space.

The Indexer creates several different indices in the Index Repository based on a set of XML documents. We do not discuss in detail all of the index structures used (e.g., inverted keyword index, inverted label index, etc.) because of space limitations. Instead, we focus our discussion on the two most important and novel index structures—the interconnection index and path index. The interconnection index allows for rapid checking of the interconnection relationship. Our path index, allows us to create first answers with higher estimated ranking. It is important to note that for each node we store an encoding that allows us to easily find the lowest common ancestor of any given pair of nodes. Basically, the encoding of a node n, is the encoding of the parent of n, augmented by the index of n among its siblings. This encoding allows the index structures to perform efficiently.

5.1 Dynamic Offline Interconnection Indexing

Checking for interconnection of nodes online is expensive. Hence, we decided at first to create a *node-interconnection index* that would store information about the interconnection relationship between each pair of nodes. This requires solving the following problem. Given a document T, for all pairs of nodes n and n' in T, determine whether n and n' are interconnected.

It is easy to see that given a document T and nodes n, n', it is possible to check whether n and n' are interconnected in time $\mathcal{O}(|T|)$, where $|T|$ is the number of nodes in T. It follows that we can check for interconnection of all pairs of nodes in T in time $\mathcal{O}(|T|^3)$. However, we improved upon this result in our XSEarch implementation by using dynamic programming. We say that nodes n and n' are *strongly-interconnected* if they are interconnected and are also labeled differently. Essentially, this corresponds to Condition 1 of interconnection from Section 3. Our algorithm is based on the following Lemma.

Lemma 5.1 (Interconnection Characterization) *Let T be a document and let n and n' be nodes in T. If n is an ancestor of n', then n and n' are interconnected if and only if the following hold:*

- *the parent of n' is strongly-interconnected with n;*
- *the child of n on the path to n' is strongly-interconnected with n'.*

If n is not an ancestor of n' and n' is not an ancestor of n, then n and n' are interconnected if and only if the following hold:

- *the parent of n' is strongly-interconnected with n;*

- *the parent of n is strongly-interconnected with n'.*

Theorem 5.2 (All Pairs Interconnection) *Let T be a document. Then it is possible to determine interconnection of all pairs of nodes in T in time $\mathcal{O}(|T|^2)$.*

Proof (Sketch). Our procedure that solves this problem is presented in Figure 4. For simplicity of exposition, we assume that the nodes in T are numbered 1 through $|T|$. In addition, we assume that this numbering was derived by a depth-first traversal of T.

We use `intercon[`i`,`j`]` to denote the boolean value of whether i and j are interconnected. The index structures that are used to efficiently execute this procedure are not specified. Note that the order that our `for` loops are evaluated (in Lines 1, 2 and in Lines 8, 9) ensures that the right hand of the assignments in Lines 6 and 13 have already been evaluated. □

We used the algorithm from Figure 4 in order to compute the interconnection values for all pairs of nodes. In the XSEarch system, we have explored the possibilities of storing the node-interconnection index in either a hashtable or a symmetric matrix. When implemented as a hashtable, the node-interconnection index contains pairs of ids of interconnected nodes. When implemented as a symmetric matrix, the node-interconnection index contains a boolean value for each pair of nodes, indicating whether they are interconnected or not. A comparison of time and space efficiency of these structures can be found in Section 6.

5.2 Dynamic Online Interconnection Indexing

Offline computation of the node-interconnection index may be expensive (see Section 6.3). In order to amortize the cost of computing this index over the queries received, we have also considered an online indexing method. When indexing online, for each pair of nodes n and n' whose interconnection must be checked (and is not yet known), we compute the section of the node interconnection index corresponding to $T_{|n,n'}$. This can be done in a fashion similar to the procedure presented in Figure 4. We use a hashtable to store the part of the index that has already been computed at any given moment. The hash table contains a boolean value for each pair of nodes whose interconnection has already been checked. The boolean value indicates whether the nodes are interconnected or not.

During query processing, usually only a small part of the node-interconnection index will be created, thus the slowdown in response time is not large. In addition, we note that queries tend to have a *locality property*. Intuitively, queries tend to be similar in the parts of the document that they must access. Therefore, even after many queries have been evaluated, it is likely for the node-interconnection index to be only partially computed. This speeds up execution time when loading the index into main memory.

5.3 Path Index

Our ranker (see Equation 1) combines three factors. One, $sim(Q,N)^\alpha$, ranks according to content and the other two rank according to structure. In cases where choosing $\alpha = 0$ yields a good ranker, the answers can be generated in the order of their ranking. When $\alpha \neq 0$, a good strategy is to generate the top answers, assuming that $\alpha = 0$, and while the user is looking at those answers, the system can generate and sort all answers for the actual value of α. In the remainder of this section, we show how to generate answers in the order of their ranking when $\alpha = 0$.

Given a document, we first find all the different paths of labels from the root to the leaves. Even very large documents tend to have a relatively small number of such paths [7]. Given two paths of labels, we can determine if they can lead to two interconnected nodes n_1 and n_2, and if so, we can determine the set of labels in the interconnection tree of n_1 and n_2. For example, consider the paths of labels $p_1 =$ `dblp.inproceedings.author` and $p_2 =$ `dblp.inproceedings.title`. These can lead to interconnected nodes n_1 and n_2 only if n_1 and n_2 share a lowest common ancestor at the second level of the tree (i.e., the `inproceedings` node). Thus, if these paths of labels lead to interconnected nodes, the relationship tree of those nodes would contain the labels `inproceedings`, `author` and `title`. As another example, consider the path $p_3 =$ `dblp.phdthesis.title`. The paths p_1 and p_3 can lead to interconnected nodes only if their lowest common ancestor is the root. Note that the relationship tree in this case would contain 5 nodes, instead of 3. Thus, a pair of interconnected nodes reachable by p_1 and p_2 would have a relationship tree of a smaller size (and hence, a higher ranking) than a pair of nodes reachable by p_1 and p_3. Our path index stores at `Path[`p_1`,`p_2`]` the labels of the nodes in the interconnection tree defined by the paths p_1 and p_2, if p_1 and p_2 can lead to interconnected nodes.

In the inverted keyword index, we store for each keyword, the paths in the documents that lead to that keyword. For each of these paths, we store the set of nodes reachable by that path. Similarly, we store for each label, the paths in the document that lead to that label and the nodes reachable by those paths. For efficiency, we actually store a *path id* instead of the entire path.

Now, before creating the answers for a query, we can determine all possible paths that may lead to keywords or labels in the query. In order to check whether a series of paths $p_1, \ldots, p_k$ can lead to k interconnected nodes, it is sufficient to verify that each pair of paths can lead to interconnected nodes [9]. This information appears in the path index. In order to compute the size of the interconnection tree determined by $p_1, \ldots, p_k$, it is sufficient to check how many distinct labels appear

XML Document	Size in KB	Number of Nodes	NII Time (ms)	IIH Time (ms)	IIM Time (ms)
Dream	146	3,360	0.625	36	29
Hamlet	281	6,635	1.11	185	114
Sigmod Record	704	21,246	2.234	1,729	1,552
Mondial	1,198	49,422	10.059	7,837	6,231

Table 1: Test documents for index generation: size in kilobytes, number of nodes and time to create indices.

among `Path`$[p_i, p_j]$ for all $1 \leq i, j \leq k$.[2] We create the query answers for the configurations with the highest ranking first.

6 Experimental Results

We performed extensive experimentation with the XSEarch system, which was implemented in Java. The experiments were carried out on a Pentium 4, with a CPU of 1.6GHZ and 2GB of RAM, running the Windows XP operating system. Note that Java can only take advantage of up to 1.46GB of RAM.

6.1 Scalability

In order to determine the scalability of XSEarch, we checked how long it takes to create an index for a document D, as a factor of the size of D and the size of the resulting index. Three different types of indices were created for a variety of XML documents: (1) **NII:** No Interconnection Index was created. We created all the other indices. This corresponds to what is created when using the dynamic online strategy for the interconnection index; (2) **IIH:** The Interconnection Index was created as a Hashtable; (3) **IIM:** The Interconnection Index was created as a symmetric Matrix.

For each of the three options, we calculated the time required in order to create the indices and the resulting size of the indices. The documents tested, along with their physical size, their number of nodes and the time it took to compute each of the three types of index generation are detailed in Table 1.

In Figure 5 we summarize the resulting index size. We consider both the size of the index on disk (i.e., zipped) and in memory (i.e., unzipped). The unzipped version of IIM is always of size proportional to the square of the number of nodes, while the zipped version of IIM, and both versions of IIH, are of size proportional to the number of pairs of interconnected nodes. Hence, in the unzipped version, it is always more space efficient to store the interconnection index as a hashtable. In the zipped version, the preferable data structure is dependent on the percentage of interconnected nodes in the document. For example, this percentage is high for the Mondial, and thus, zipped IIH turns out to be smaller than zipped IIM in this

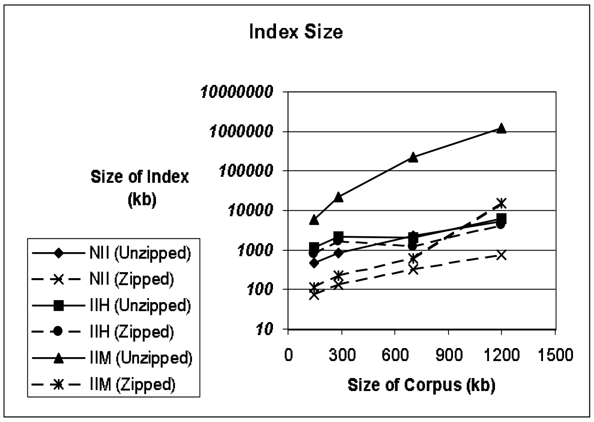

Figure 5: Size of index, zipped and unzipped.

case. Observe that the time needed and the index size grow polynomially as functions of the document size.

6.2 Query Execution Time

In order to check how query execution time is affected by the semantics of the query (i.e., all-pairs or star) and the type of interconnection index used, we generated 1000 random queries for the Sigmod Record document. These queries had at most 3 required search terms and at most 3 optional search terms. The keywords and labels in the queries were drawn randomly from the set of keywords and labels in the Sigmod Record. We executed the queries to determine execution time using either a hashtable or a matrix as the interconnection index. We ran the queries under both all-pairs and star semantics.

In Figure 6, a histogram of the number of milliseconds needed to process a query is presented. Note that the number of queries is on logarithmic scale. Observe that querying using a hashtable or a matrix yields similar results. Processing queries under star semantics tends to be slower than processing under all-pairs semantics, since the query result is often larger. However, in all cases, over 80% of the queries ran in under 10 milliseconds and over 97% of queries ran in under 100 milliseconds. The average run time for queries with all-pairs semantics was about 35 milliseconds and with star semantics was about 634 milliseconds.

[2]Actually, Condition 2 of interconnection requires us to slightly modify this calculation.

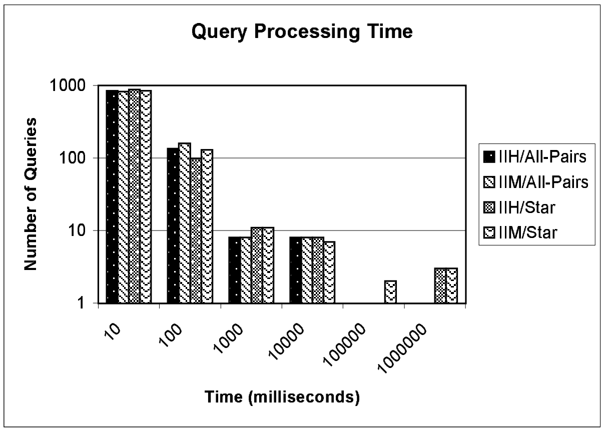

Figure 6: Histogram of processing time for random queries using various semantics and index structures.

6.3 Indexing Strategy

We considered creating the interconnection index offline in its entirety, or online incrementally. The affect of the indexing technique chosen on the size of the index and the query execution time was studied. Offline computation tends to be expensive. For some huge documents, such as the full version of the DBLP (121MB), Java's memory constraints did not allow us to create the index offline in its entirety. In addition, the index is usually large. Hence, loading the index during query processing is costly. When the interconnection index is created incrementally online, we compute those parts of the index that are (1) necessary for a given query and (2) have not yet been computed. Obviously, this increases query computation time.

We ran the random queries from Section 6.2 under all-pairs semantics, while dynamically creating the index during the query processing. More than 50% of the queries were processed in under 10 milliseconds and over 85% of the queries were processed in under 1 second. There were less than 50 queries that took over 10 seconds. After processing all 1000 random queries, 0.75% of all pairs of nodes were checked for interconnection. We postulate that for "real" queries (as opposed to randomly generated queries) even less of the interconnection index would have been created.

6.4 Example Scenario

Performing extensive user studies to determine the the precision and recall of XSEarch was beyond the scope of this paper. However, the following example can shed some light on these questions.

Suppose that we wish to find papers written by `Buneman` that contain the keyword `database` in the title. We present the XQuery query that expresses these requirements. Note that this query is complicated for a naive user and would have to be slightly changed for it to be applied to the Sigmod Record document.

```
<answers> {
   for $r in
        document("dblp.xml")//article |
        document("dblp.xml")//inproceedings
   for $a in $r//author
   for $t in $r//title
   where contains(string($a), "Buneman") and
   contains(lower-case(string($t)),
            "database")
   return {$r}
} </answers>
```

A naive user could attempt to retrieve the data required from both the Sigmod Record and the DBLP using XSEarch. However, the user may not be familiar with the exact ontology of the document, or might not be sure exactly which keywords to look for. Hence, we considered three variations of this query:

$$Q_{\text{kw}}(+\texttt{:Buneman},\ +\texttt{:database})$$
$$Q_{\text{tag}}(+\texttt{author:},\ +\texttt{title:})$$
$$Q_{\text{kw+tag}}(+\texttt{author:Buneman},\ +\texttt{title:database})$$

We ran each of the three queries on both the Sigmod Record and on a large representative sample of DBLP. Both all-pairs semantics and star semantics yield the same results. We compared the XSEarch results to:

- **Correct Results:** This is the set of actual results that the user wished to find. They were calculated using the Galax[3] XQuery engine and a query similar to the one above.
- **Naive Results:** This is the set of results that would have been retrieved if we only required that the nodes in each result satisfy the query, but did not require that the nodes in a result be interconnected. This corresponds to using the relationship R that contains every pair of nodes in the tree.

Both XSEarch and the naive approach yielded perfect recall. However, the precision, i.e., the number of correct answers that were returned relative to the number of answers that were returned, differed in each case. The precision of XSEarch and of the naive method is presented in Figure 7. Note that XSEarch always outperforms the naive approach.

Note that for DBLP, the precision of XSEarch was actually quite low, and only slightly outperformed the naive approach. For example, the precision of XSEarch for $Q_{\text{kw+tag}}$ over DBLP was approximately 0.02, versus approximately 0.01 for the naive approach. This is because title and author nodes that belong to publications of different types (e.g., title of journal and author of book) are interconnected. For cases such as this, in which interconnection does not sufficiently capture actual node relationships, proper ranking of results is of utmost importance.

[3]`http://db.bell-labs.com/galax`

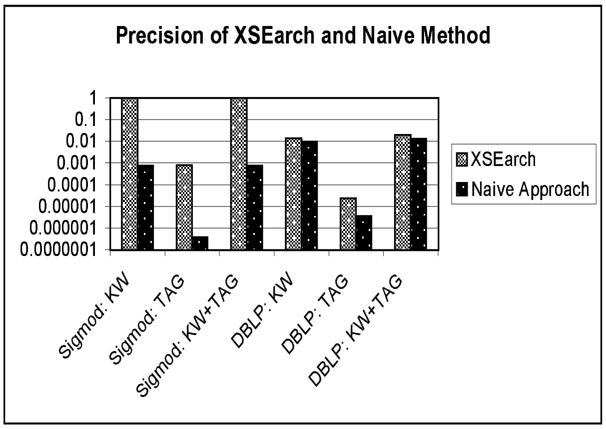

Figure 7: Precision of Q_{kw}, Q_{tag} and $Q_{\text{kw+tag}}$ for Sigmod Record and DBLP.

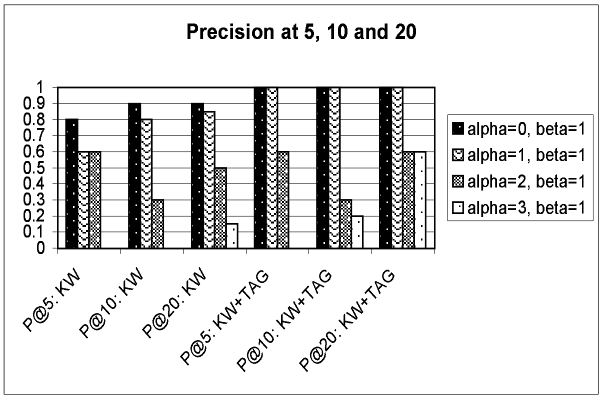

Figure 8: P@5, P@10 and P@20 of Q_{kw} and $Q_{\text{kw+tag}}$ for DBLP.

In order to test our ranker, for each of $n = 5, 10, 20$ we checked the ratio of correct answers in the first n answers returned to the number n. This is called precision at 5, precision at 10 and precision at 20, also written as P@5, P@10 and P@20. For the query Q_{tag} the precision in all these cases was always 0 since the keywords were not supplied, and thus, were not taken into consideration for the ranking. For the Sigmod Record, the overall precision of Q_{kw} and $Q_{\text{kw+tag}}$ was perfect, and hence it was perfect at 5, 10 and 20. For the remaining cases, i.e., Q_{kw} and $Q_{\text{kw+tag}}$ on DBLP, the precision at 5, 10 and 20 for a constant β and different values of α are depicted in Figure 8. The value for γ was irrelevant since no results contained nodes in an ancestor-descendent relationship. Interestingly, for these cases it turns out that if tags are not supplied in the query, it is better to give a larger weight to the size of the relationship tree, i.e., to β, in comparison with α. In any case, since the corpus contained mostly labels and not keywords, giving a large weight to α degraded the results. In documents that contain a larger ratio of keywords to labels, a larger value for α could improve the ranking of results. Further experimentation is needed to find optimal values for α, β and γ.

7 Related Work and Conclusion

Numerous query languages for XML have been developed. Recently, interest has arisen in techniques for "flexible querying" of XML. For example, the XQuery working group is considering how to add full-text search features and ranking to XQuery [5]. Such capabilities have already been added to various XML query languages. [10] extends XML-QL with keyword search and presents performance experiments. XIRQL [11] is an extension of XQL that supports vague predicates, weighting of terms and minimal structural abstracting (e.g., abstraction of differences between attributes and elements). The XXL search engine [15] has an SQL-like syntax, extended with ranking and ontological knowledge for similarity metrics. On the whole, these languages are not suitable for naive user, since the query syntax is always complex.

The EquiX [8] language is a simple extension of a search engine for XML documents. However, EquiX can only deal with documents that have a DTD. In [4], another search language for XML was proposed. Their language consists of fragments of XML documents, and they only require approximate matching of the queries to the documents. However, their query answers consist of entire documents, instead of document fragments. In addition, they do not require any semantic relationship between the parts of the document that match a given query. In [2, 13], it is suggested to rank query answers according to the distance in the document between the different document elements that satisfy a query. Closer elements would receive a higher ranking. We also use this measure for our ranking, but it is only one of several measures. In [13], efficient algorithms to compute the top k answers are presented. However, these algorithms are based on the assumption that each document has a schema, which may not necessarily hold. A theoretical treatment of the problem of flexibly matching a query to a document was presented in [14]. However, their approach did not include keyword searching.

A recent related work is the XRANK system [12] for keyword searching in XML documents. XRANK has a ranking mechanism and it returns document fragments as answers. In XRANK, there is no distinction between keywords and labels, and each keyword of an XRANK query is matched against every word of the document (even if that word is a label). An answer to an XSEarch query is also an answer or some part of an answer to the XRANK query that consists of the same keywords and labels, but the converse is not necessarily true. Actually, XRANK may return answers with parts that are semantically unrelated, as in Example 1.1. XRANK ranks the elements of an XML document by generalizing the Page-Rank algorithm of Google [3]. It ranks the answers to a query by combining the ranking of elements with keyword proximity. The notion of proximity in XRANK means that the children of an element must be in the "right order" if that element should be ranked highly as an answer. For example, if a `paper` element has `title` as its first child and `author` as its last child, with all the `section` elements in between, then that `paper` element will get a low rank, even if the query has a keyword from the title and a keyword from author's name. In XSEarch, proximity is included in the ranking formula in terms of the size of the relationship tree and thus, it is not affected by the order of children. XSEarch employs more information-retrieval techniques than XRANK, namely, *tfidf* and similarity between the query and the document. The element ranking used in XRANK can also be incorporated in XSEarch, but its utility is not clear. It seems to be useful in DBLP, where references between elements indicate importance. However, what is the significance of a large number of references in a document about geographical data (e.g., Mondial), where references are between neighboring countries?

The main contribution of this paper is in laying the foundations for a semantic search engine over XML documents. XSEarch returns semantically related fragments, ranked by estimated relevance. Our system is extensible, and can easily accommodate different types of relationships between nodes. We have shown that it is possible to combine these qualities with an efficient, scalable and modular system. Thus, XSEarch can be seen as a general framework for semantic searching in XML documents.

Acknowledgements

Jonathan Mamou is grateful to Marie-Christine Rousset for helpful discussions. This work was supported by the Israel Science Foundation (Grant 96/01).

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[2] M. Barg and R. Wong. Structural proximity searching for large collections of semi-structured data. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management*, pages 175–182, Atlanta (Georgia, USA), Nov. 2001. ACM Press.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1–7):107–117, 1998.

[4] D. Carmel, Y. Maarek, Y. Mass, N. Efraty, and G. Landau. An extension of the vector space model for querying XML documents via XML fragments. In *ACM SIGIR 2002 Workshop on XML and Information Retrieval*, Tampere (Finland), Aug. 2002.

[5] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML query use cases. `http://www.w3.org/TR/2002/WD-xmlquery-use-cases-20020816`.

[6] T. Chinenyanga and N. Kushmerick. An expressive and efficient language for XML information retrieval. *Journal of the American Society for Information Science and Technology*, 53(6), 2002.

[7] B. Choi. What are real dtds like? In *The Fifth International Workshop on Web and Databases (WebDB)*, Madison (Wisconsin, USA), June 2002.

[8] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. EquiX: A search and query language for XML. *Journal of the American Society for Information Science and Technology*, 53(6):454–466, 2002.

[9] S. Cohen, Y. Kanza, and Y. Sagiv. Generating relations from XML documents. In *Proc 9th International Conference on Database Theory*, Siena (Italy), Jan. 2003. Springer-Verlag.

[10] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *The International Journal of Computer and Telecommunications Networking*, 33(1):119–135, June 2000.

[11] N. Fuhr and K. Großjohann. XIRQL: a query language for information retrieval in XML documents. In *Proc. 24th International ACM SIGIR Conference on Research and Development in Information Retreival*, pages 172–180, New Orleans (Louisiana, USA), Sept. 2001. ACM Press.

[12] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *Proc. 2003 ACM SIGMOD International Conference on Management of Data*, San Diego (California), June 2003.

[13] V. Hristidis and Y. P. A. Balmin. Keyword proximity search on XML graphs. In *Proc. 19th International Conference on Data Engineering*, Bangalore (India), Mar. 2003.

[14] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proc. 20th Symposium on Principles of Database Systems*, pages 40–51, Santa Barbara (California, USA), May 2001. ACM Press.

[15] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *Proc. 8th International Conference on Extending Database Technology*, pages 477–495, Prague (Czech Republic), March 2002. Springer-Verlag.